

NetworkX Overview

Introduction

NetworkX is built to allow easy creation, manipulation and measurement on large graph theoretic structures which we call networks. The graph theory literature defines a graph as a set of nodes(vertices) and edges(links) where each edge is associated with two nodes. In practical settings there are often properties associated with each node or edge. These structures are fully captured by NetworkX in a flexible and easy to learn environment.

Graphs

The basic object in NetworkX is a Graph. The simplest type of Graph has undirected edges, does not allow multiple edges between nodes and does not allow self-loops (edges from a node to itself). This corresponds to a “simple graph” in the literature. The Graph class contains this data structure. Other classes allow directed graphs (DiGraph), self-loops and multiple edges, and the storage of properties for edges and nodes (XGraph).

We will discuss the Graph class first.

Empty graphs can be created most easily using the Graph class:

```
>>> import networkx
>>> G=networkx.Graph()
```

At this point, the graph G has no nodes or edges and has the name “No Name”. The name can be set through the variable G.name or through an optional argument such as

```
>>> G=networkx.Graph(name="I have a name!")
>>> print G.name
I have a name!
```

Graph Manipulation:

Basic graph manipulation is provided through methods to add or delete nodes or edges.

```
>>> G.add_node(1)
>>> G.delete_node(1)
>>> G.add_edge(1,2)
>>> G.delete_edge(1,2)
>>> G.delete_edge(1,2)
```

```
>>> G.delete_edge(2,3)
>>> G.delete_edge(3,1)
```

If a node is deleted, all edges associated with that node are also deleted. If an edge is added connecting a node that does not yet exist, that node is added when the edge is added.

Arguments to `add_node/delete_node` methods may be single entries or lists of entries (but not tuples of entries as the tuple is considered to be the name of the node). Arguments to the `add_edge/delete_edge` methods may be two entries (the names of the two nodes associated with the edge), a single 2-tuple entry or a list of 2-tuple entries.

More complex manipulation is available through the methods:

```
G.add_path([list of nodes]) - add edges to make this ordered path.
G.add_cycle([list of nodes]) - same as path, but connect ends.
G.clear() - delete all nodes and edges.
G.copy() - a "shallow" copy of the graph just like dict.copy()
G.subgraph([list of nodes]) - the subgraph of G containing those nodes.
```

In addition, the following functions are available:

```
union(G1,G2) -combine graphs without any edges connecting the two.
compose(G1,G2) -combine graphs identifying nodes with same names.
complement(G) -the graph complement (edges exist if they don't in G).
create_empty_copy(G) - return an empty copy of the same graph class.
```

You should also look at the graph generation routines in `networkx.generators`

Methods:

Some properties are tied more closely to the data structure and can be obtained through methods as well as functions. These are:

- `G.order()`
- `G.size()`
- `G.nodes()`: a list of nodes in G.
- `G.nodes_iter()`: an iterator over the nodes of G.
- `G.has_node(v)`: check if node v in G (returns True or False).
- `G.edges()`: a list of 2-tuples specifying edges of G.
- `G.edges_iter()`: an iterator over the edges (as 2-tuples) of G.
- `G.has_edge(u,v)`: check if edge (u,v) in G (returns True or False).
- `G.neighbors(v)`: list of the neighbors of node v (outgoing if directed).
- `G.neighbors_iter(v)`: an iterator over the neighbors of node v.
- `G.degree()`: list of the degree of each node.
- `G.degree_iter()`: an iterator over the degrees of the nodes in G.

Argument syntax and options are the same as for the functional form.

Node/Edge Property Functions

Two functions exist for each node property. With no suffix on the function name, the function returns an a list or dictionary with property values for each node. The suffix “_iter” in a function name signifies that an iterator will be returned.

You should be careful to avoid changing the graph when using an iterator. Also, some operations (like len()) need a list instead of an iterator. If in doubt, use the regular form of the function. We think you will still most often use the standard iterator functions because the idiom:

*for v in G.nodes():
 do something with v
is hard to avoid.*

In addition, an optional argument “with_labels” signifies whether the returned values should be identified by node or not. If “with_labels=False”, only property values are returned in either iterator or list form. If “with_labels=True”, a dict of property values keyed by node is returned.

The following examples use the “triangles” function which returns the number of triangles a given node belongs to.

Example usage

```
>>> G=networkx.complete_graph(4)
>>> print G.nodes()
[0, 1, 2, 3]
>>> print networkx.triangles(G)
[3, 3, 3, 3]
>>> print networkx.triangles(G,with_labels=True)
{0: 3, 1: 3, 2: 3, 3: 3}
```

Properties for specific nodes

Many node property functions return property values for either a single node, a list of nodes, or the whole graph. The return type is determined by an optional input argument.

1. By default, values are returned for all nodes in the graph.
2. If input is a list of nodes, a list of values for those nodes is returned.
3. If input is a single node, the value for that node is returned.

Node v is special for some reason. We want to print info on it.

```
>>> v=1
>>> print "Node %s has %s triangles."%(v,networkx.triangles(G,v))
Node 1 has 3 triangles.
```

Maybe you need a polynomial on t ?

```
>>> t=networkx.triangles(G,v)
>>> poly=t**3+2*t-t+5
```

Get triangles for a subset of all nodes.

```
>>> vlist=range(0,4)
>>> for (v,t) in networkx.triangles(G,vlist,with_labels=True).items():
...     print "Node %s is part of %s triangles"%(v,t)
Node 0 is part of 3 triangles.
Node 1 is part of 3 triangles.
Node 2 is part of 3 triangles.
Node 3 is part of 3 triangles.
```